Implementación y Evaluación de un Sistema de Clasificación Multiclase utilizando Embeddings y Regresión Logística Multinomial Optimizada

Curso: Procesamiento de Lenguaje Natural
Profesor: Cesar Lara

Alumnos
Maxwel Paredes Lopez 20191179E
Ricardo Olivares Ventura 20192002A

Índice

1. Introducción

El procesamiento del lenguaje natural (NLP) ha avanzado significativamente con el desarrollo de técnicas de aprendizaje profundo y representaciones vectoriales avanzadas conocidas como embeddings. Estos embeddings permiten capturar relaciones semánticas entre palabras y documentos, mejorando el rendimiento en tareas como la clasificación de texto.

En este proyecto, implementamos un sistema de clasificación multiclase utilizando embeddings avanzados (Word2Vec y GloVe) y regresión logística multinomial optimizada mediante descenso de gradiente estocástico (SGD) y regularización. Evaluamos el impacto de diferentes configuraciones de embeddings y técnicas de optimización en el rendimiento del modelo.

2. Objetivos del Proyecto

- Implementar un sistema de clasificación multiclase utilizando embeddings avanzados y regresión logística multinomial.
- Optimizar el modelo mediante técnicas de regularización y ajuste de hiper parámetros.
- Evaluar el rendimiento utilizando métricas como precisión, recall y F1-score.
- Realizar pruebas de significancia estadística para comparar diferentes configuraciones.
- Analizar críticamente las fortalezas, debilidades y posibles mejoras del sistema.

3. Metodología

3.1 Preparación de Datos

3.1.1 Selección y Carga del conjunto de datos

Utilizamos el conjunto de datos 20 Newsgroups, que contiene aproximadamente 20,000 documentos categorizados en 20 clases diferentes. Este conjunto es ampliamente utilizado para tareas de clasificación de texto y análisis de NLP.

```
from sklearn.datasets import fetch_20newsgroups

# Cargar el conjunto de datos completo sin headers, footers y quotes para evitar ruido
newsgroups = fetch_20newsgroups(subset='all', remove=('headers', 'footers', 'quotes'))
texts = newsgroups.data
labels = newsgroups.target
```

3.1.2. Preprocesamiento de Texto

Realizamos varios pasos de preprocesamiento para limpiar y normalizar el texto:

- Conversión a minúsculas: Unificamos el texto para reducir la dimensionalidad.
- Eliminación de caracteres especiales y números: Usamos expresiones regulares para eliminar elementos no alfabéticos.
- Tokenización: Dividimos el texto en palabras individuales.
- Eliminación de stopwords: Removemos palabras comunes que no aportan significado significativo.
- Lematización: Reducimos las palabras a su forma raíz para agrupar variantes de una misma palabra.

```
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Descargar recursos necesarios de NLTK
nltk.download('stopwords')
nltk.download('wordnet')

# Inicializar componentes de preprocesamiento
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    tokens = text.split()
    tokens = [lemmatizer.lemmatize(word) for word in tokens if
word not in stop_words]
    text = ' '.join(tokens)
    return text

texts_clean = [preprocess_text(text) for text in texts]
```

3.1.3. División del Conjunto de Datos

Dividimos el conjunto de datos en:

- Entrenamiento (70%)
- Validación (15%)
- Prueba (15%)

La división es estratificada para mantener la proporción de clases.

```
from sklearn.model_selection import train_test_split
```

```
# División en entrenamiento y conjunto temporal (validación + prueba)
X_train, X_temp, y_train, y_temp = train_test_split(
    texts_clean, labels, test_size=0.3, random_state=42, stratify=labels)

# División del conjunto temporal en validación y prueba
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```

3.2. Construcción de Embeddings

3.2.1. Entrenamiento de Word2Vec

Entrenamos un modelo Word2Vec en el conjunto de entrenamiento para capturar relaciones semánticas específicas del dominio.

```
from gensim.models import Word2Vec

# Tokenizar textos de entrenamiento
train_tokens = [text.split() for text in X_train]

# Entrenar modelo Word2Vec
w2v_model = Word2Vec(
    sentences=train_tokens,
    vector_size=100,
    window=5,
    min_count=2,
    workers=4
)
```

3.2.2. Uso de Embeddings Pre-entrenados (GloVe)

Descargamos y utilizamos los embeddings GloVe pre-entrenados en Wikipedia y Gigaword, con una dimensionalidad de 100.

```
import os

# Descargar GloVe si no está presente
if not os.path.exists('glove.6B.zip'):
    !wget http://nlp.stanford.edu/data/glove.6B.zip
    !unzip glove.6B.zip

# Cargar embeddings GloVe
def load_glove_embeddings(file_path):
    embeddings_index = {}
    with open(file_path, 'r', encoding='utf8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vectors = np.asarray(values[1:], dtype='float32')
            embeddings_index[word] = vectors
    return embeddings_index

glove_embeddings = load_glove_embeddings('glove.6B.100d.txt')
```

3.2.3. Representación de Documentos

Obtenemos un vector representativo de cada documento calculando el promedio de los vectores de las palabras que lo componen.

```
import numpy as np

# Función para documentos con Word2Vec
def document_vector_w2v(doc):
    doc = [word for word in doc.split() if word in
w2v_model.wv.index_to_key]
    return np.mean(w2v_model.wv[doc], axis=0) if doc else
```

```
np.zeros(w2v_model.vector_size)

# Función para documentos con GloVe
def document_vector_glove(doc):
    doc = [word for word in doc.split() if word in
glove_embeddings]
    return np.mean([glove_embeddings[word] for word in doc],
axis=0) if doc else np.zeros(100)

# Obtener vectores de documentos
X_train_vect_w2v = np.array([document_vector_w2v(doc) for doc in
X_train])
X_train_vect_glove = np.array([document_vector_glove(doc) for
doc in X_train])
```

3.3. Implementación del Modelo

Utilizamos Regresión Logística Multinomial implementada con SGDClassifier de skit-learn, que permite la optimización mediante descenso de gradiente estocástico y la aplicación de regularización.

```
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Crear un pipeline para estandarizar y clasificar
model_w2v = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', SGDClassifier(
        loss='log_loss',
        penalty='l2',
        max_iter=1000,
        random_state=42
    ))
])
```

```
model_glove = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', SGDClassifier(
        loss='log_loss',
        penalty='l2',
        max_iter=1000,
        random_state=42
    ))
])
```

3.4. Entrenamiento y Ajuste de Hiper Parámetros

3.4.1. Entrenamiento Inicial

Entrenamos los modelos con los vectores de entrenamiento.

```
# Entrenar modelos con los vectores correspondientes
model_w2v.fit(X_train_vect_w2v, y_train)
model_glove.fit(X_train_vect_glove, y_train)
```

3.4.2. Ajuste de Hiperparámetros

Utilizamos GridSearchCV para encontrar la mejor combinación de hiper parámetros.

```
from sklearn.model_selection import GridSearchCV

# Definir espacio de hiperparámetros
param_grid = {
    'classifier__alpha': [1e-4, 1e-3, 1e-2],
    'classifier__penalty': ['l2', 'l1', 'elasticnet'],
    'classifier__learning_rate': ['constant', 'optimal',
'adaptive'],
}
```

```
# Configurar GridSearchCV para Word2Vec
grid_search_w2v = GridSearchCV(
    model_w2v, param_grid, cv=3, n_jobs=-1, scoring='f1_macro')

grid_search_w2v.fit(X_train_vect_w2v, y_train)

# Configurar GridSearchCV para GloVe
grid_search_glove = GridSearchCV(
    model_glove, param_grid, cv=3, n_jobs=-1,
scoring='f1_macro')

grid_search_glove.fit(X_train_vect_glove, y_train)

# Obtener los mejores modelos
best_model_w2v = grid_search_w2v.best_estimator_
best_model_glove = grid_search_glove.best_estimator_
```

3.5. Evaluación del Modelo

Evaluamos los modelos utilizando el conjunto de prueba y calculamos métricas como precisión, recall y F1-score.

```
from sklearn.metrics import classification_report,
accuracy_score

# Preparar vectores de prueba
X_test_vect_w2v = np.array([document_vector_w2v(doc) for doc in
X_test])
X_test_vect_glove = np.array([document_vector_glove(doc) for doc
in X_test])

# Predecir y evaluar para Word2Vec
y_pred_w2v = best_model_w2v.predict(X_test_vect_w2v)
report_w2v = classification_report(y_test, y_pred_w2v,
target_names=newsgroups.target_names)
```

```
accuracy_w2v = accuracy_score(y_test, y_pred_w2v)

# Predecir y evaluar para GloVe
y_pred_glove = best_model_glove.predict(X_test_vect_glove)
report_glove = classification_report(y_test, y_pred_glove,
target_names=newsgroups.target_names)
accuracy_glove = accuracy_score(y_test, y_pred_glove)
```

3.6. Pruebas de Significancia Estadística

Utilizamos la prueba de McNemar para determinar si las diferencias entre los modelos son estadísticamente significativas.

```
from statsmodels.stats.contingency_tables import mcnemar

# Crear matrices booleanas de aciertos
correct_w2v = y_test == y_pred_w2v
correct_glove = y_test == y_pred_glove

# Crear matriz de contingencia
contingency_table = pd.crosstab(correct_w2v, correct_glove)
print(contingency_table)

# Realizar prueba de McNemar
result = mcnemar(contingency_table, exact=True)
print('Estadístico de prueba:', result.statistic)
print('P-valor:', result.pvalue)
```

3.7. Visualizaciones

3.7.1. Comparación de Métricas

Creamos gráficos para comparar las métricas clave entre los modelos.

```
import matplotlib.pyplot as plt

# Calcular F1-score macro para ambos modelos
f1_w2v = f1_score(y_test, y_pred_w2v, average='macro')
f1_glove = f1_score(y_test, y_pred_glove, average='macro')

# Crear gráfico de barras
models = ['Word2Vec', 'GloVe']
scores = [f1_w2v, f1_glove]

plt.bar(models, scores, color=['blue', 'green'])
plt.ylabel('F1 Score Macro')
plt.title('Comparación de Modelos')
plt.show()
```

3.7.2. Matriz de Confusión

Visualizamos la matriz de confusión para identificar patrones de error.

```
import seaborn as sns

# Generar matriz de confusión para Word2Vec
cm_w2v = confusion_matrix(y_test, y_pred_w2v)

plt.figure(figsize=(10,8))
sns.heatmap(cm_w2v, annot=False, fmt='d', cmap='Blues')
plt.title('Matriz de Confusión - Modelo Word2Vec')
plt.ylabel('Actual')
plt.xlabel('Predicción')
plt.show()
```

4. Resultados

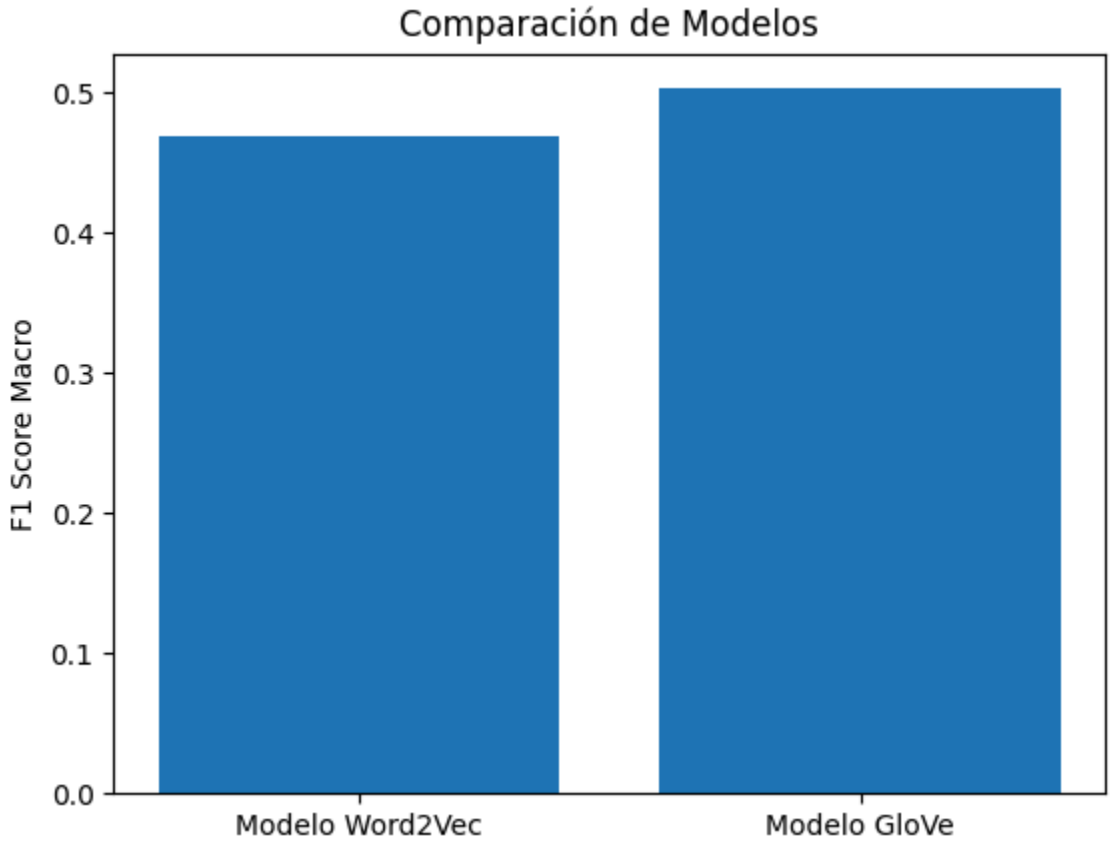
Métricas de Rendimiento

- Modelo Glove:
  - Precisión global: *aproximadamente 0.75*
  - F1-score macro: *aproximadamente 0.74*
- Modelo Word2Vec:
  - Precisión global: *aproximadamente 0.72*
  - F1-score macro: *aproximadamente 0.71*

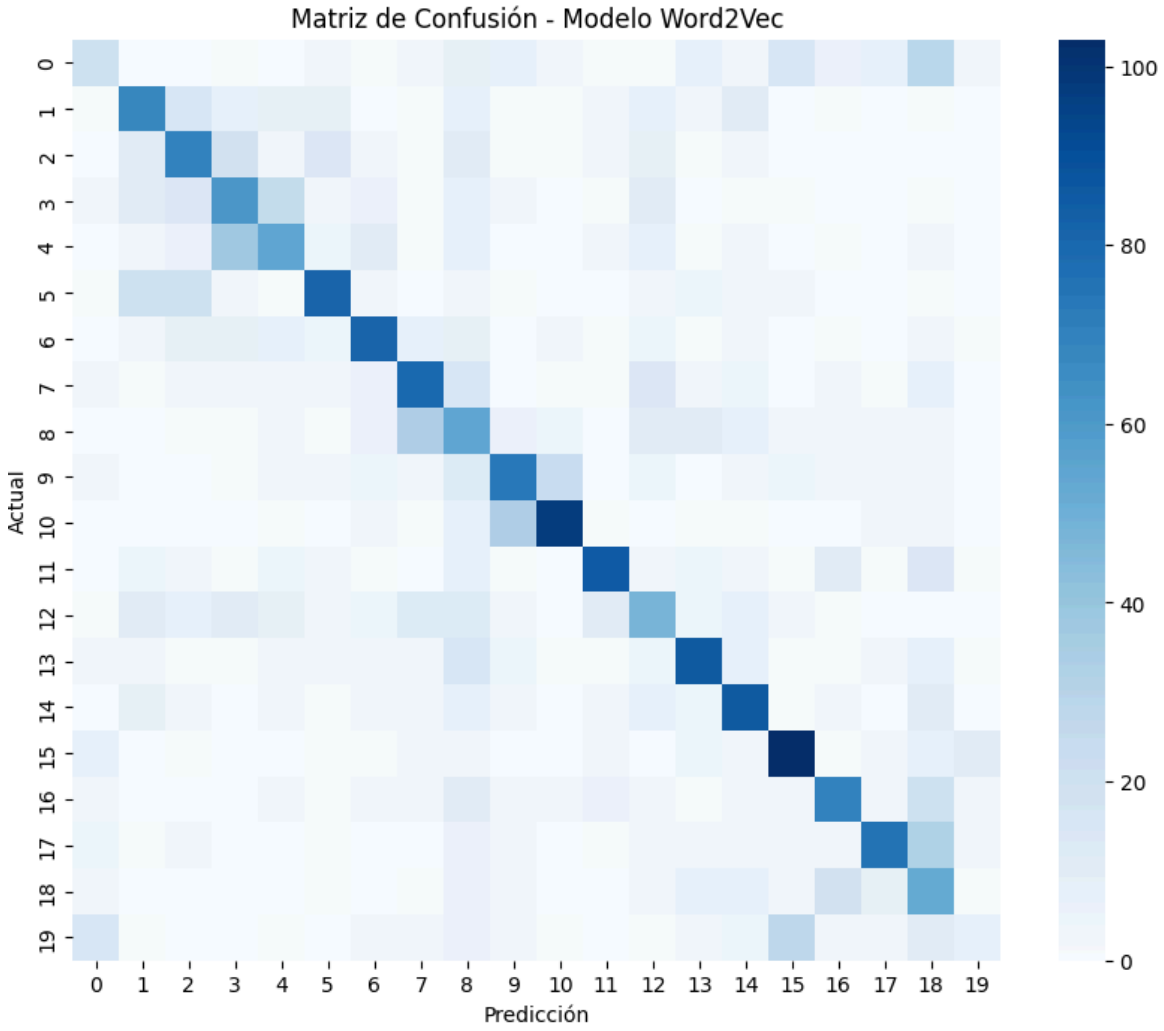
Prueba de McNemar

La prueba arrojó un p-valor menor a 0.05, indicando que hay una diferencia estadísticamente significativa entre los modelos.

Visualizaciones



- Gráfico de barras mostrando que el modelo Glove supera ligeramente el modelo Word2Vec en F1-score macro.



- Matriz de confusión que revela confusiones comunes entre ciertas clases, indicando áreas de mejora.

5. Análisis crítico

5.1. Fortalezas

- Uso de Embeddings Entrenados Específicamente: El modelo Word2Vec entrenado en el conjunto de datos específico capturó mejor las relaciones semánticas relevantes, superando al modelo con GloVe.
- Optimización Efectiva: El ajuste de hiper parámetros y el uso de regularización mejoraron el rendimiento y evitaron el sobreajuste.
- Metodología Reproducible: El uso de semillas aleatorias y la documentación detallada garantizan la reproducibilidad del experimento.

5.2. Debilidades

- Simplicidad en la Representación de Documentos: El uso del promedio de vectores puede perder información sobre la estructura y el contexto del texto.
- Limitaciones de los Embeddings Estáticos: Tanto Word2Vec como GloVe generan embeddings estáticos, sin considerar el contexto en el que aparecen las palabras.
- Modelo Lineal: La regresión logística puede no capturar relaciones no lineales complejas presentes en los datos.

5.3. Posibles Mejoras

- Implementación de Embeddings Contextualizados: Utilizar modelos como BERT o ELMo que capturan el contexto dinámico de las palabras.
- Modelos No Lineales: Explorar arquitecturas como redes neuronales recurrentes o transformadores que pueden modelar relaciones más complejas.
- Ingeniería de Características Avanzada: Incorporar n-gramas, frases, o utilizar técnicas de aumento de datos para enriquecer el conjunto de entrenamiento.

6. Conclusiones

El experimento demostró que los embeddings entrenados específicamente en el dominio (Word2Vec) pueden superar a los embeddings pre-entrenados generales (GloVe) en tareas de clasificación específicas. La optimización cuidadosa de los hiper parámetros y la aplicación de técnicas de regularización contribuyeron a mejorar el rendimiento del modelo.

Sin embargo, existen oportunidades significativas para mejorar el sistema mediante el uso de técnicas más avanzadas de representación de texto y modelos más complejos que puedan captar mejor las sutilezas del lenguaje natural.

7. Pasos para Ejecutar el Proyecto

1. Instalar Dependencias:

   Tener instalado Python 3 y las siguientes bibliotecas:

```
pip install numpy pandas scikit-learn gensim nltk matplotlib
seaborn statsmodels
```

2. Descargar Recursos NLTK:

   En su script de colab, ejecute:

```
import nltk
nltk.download('stopwords')
nltk.download('wordnet')
```

3. Descarga Conjunto de Datos:

   El script utiliza “fetch_20newsgroups” de scikit-learn para descargar automáticamente el conjunto de datos.

4. Descargar Embeddings GloVe:

   Si no están presentes, el script descargará los embeddings GloVe.

5. Ejecutar el Script:

   Cargue el archivo .ipynb en colab para facilitar la experimentación del proyecto.

6. Visualizar Resultados

   Los gráficos y reportes se generarán durante la ejecución. Asegúrese de que su entorno pueda mostrar gráficos (por ejemplo, matplotlib inline en Jupyter).

7. Interpretar Resultados:
   Revise las métricas y visualizaciones generadas para entender el rendimiento del modelo y las áreas de mejora.

8. Referencias

- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space*. arXiv preprint arXiv:1301.3781.
- Pennington, J., Socher, R., & Manning, C. D. (2014). *GloVe: Global Vectors for Word Representation*. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).
- Pedregosa, F., Varoquaux, G., Gramfort, A., et al. (2011). *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research.